Inferring Types by Mining Class Usage Frequency from Inline Caches

Nevena Milojković, Clément Bera, Mohammad Ghafari, Oscar Nierstrasz

Inferring Types by Mining Class Usage Frequency from Inline Caches

Nevena Milojković
SCG, University of Bern
nevena@inf.unibe.ch

Clément Béra
RMOD - INRIA Lille Nord Europe
clement.bera@inria.fr

Mohammad Ghafari
SCG, University of Bern
ghafari@inf.unibe.ch

Oscar Nierstrasz
SCG, University of Bern
scg.unibe.ch

Abstract

Dynamically typed languages allow developers to write more expressive source code, but their lack of static information about types of variables increases the complexity of a program. Static type information about types of variables facilitates program comprehension and maintenance.

Simple type inference algorithms suffer from the problem of false positives or negatives, thus complex approaches are required to avoid this problem.

We propose a simple heuristic that uses easily accessible run-time information about the usage of each class as a receiver type for a message send. This frequency serves as a proxy for the likelihood that a run-time type of the variable is that class and it is used to promote the correct type towards the top of the list of possible types for a variable.

Our evaluation of a proof-of-concept prototype implemented in Pharo Smalltalk shows that our heuristic is reasonably precise to detect correct types on average in 65.92% to 82.83% of cases.

1. Introduction

The lack of static type information in dynamically typed languages hampers program comprehension process [22], [2]. This is very important since 70% of development time is devoted to program maintenance [7]. Even though dynamically typed languages increase developer's productivity in writing source code [22, 26], having type information available facilitates program comprehension [17, 19, 21], *e.g.,* it decreases software maintenance time.

The problem of inferring types of variables in dynamically typed languages has been long known [1, 3, 4, 10, 23, 25, 27–29, 31, 33]. The usual approach is to statically track variable assignments and the set of messages sent to a variable, *i.e.,* to track down all messages sent to the variable of interest and then to deduce which classes understand them, either because they implement those methods, or inherit them from a superclass [27]. Other approaches produce their results by exploiting dynamic information [4, 10, 25, 28]. However, in order to acquire that information, a complete, running program must be available, usually instrumented to log the types of variable [4, 25]. In a majority of the algorithms, this kind of analysis introduces an overhead to program execution.

Nowadays many virtual machines for dynamic languages include Just-in-Time compilers that use inline caches [11, 20] to achieve high performance. That is, while the program runs, instead of interpreting the bytecode, the virtual machine translates it to machine code, and executes the machine code version. Each message send in the machine code version has its own look-up cache, called an inline cache. Such caches contain type information about the receiver of a message send, which could be easily exploited in order to improve current tools for program comprehension. We believe that this information collected during execution of *any* program written in the same language would add productively to the statically collected knowledge used for inferring variable's type. As dynamic information has been read from the virtual machine, no instrumentation is required, and minimal overhead is introduced.

We present an approach for collecting run-time information about the receiver's type from inline caches with minimal execution overhead, and we use this information to sort classes that represent a possible variable type based on the

frequency of their usage. We propose that the frequency of class usage as the type of a receiver can serve as a reliable proxy to identify the type of a variable at run time.

We have implemented a proof-of-concept for Pharo Smalltalk, a highly reflective dynamically typed language [16]. We have used this implementation to evaluate our claim. The results show that the implemented heuristic is reasonably precise for more than two thirds of the variables. The results of our approach were compared to the EATI (Eco-aware type inference) technique [30]. EATI collects information on how many times each selector has been sent to an instance of a certain class. Based on this frequency, it orders possible types for a variable. With fewer requirements, we have obtained better results, *i.e.,* our heuristic correctly infers types for 48% more variables than EATI.

Structure of the Paper. We discuss related work in section 2. Section 3 explains the virtual machine used for dynamic data collection. Next we define the used terminology and the implemented heuristic in section 4. Section 5 shows results of the evaluation of the prototype. We then describe potential threats to validity in section 6 before concluding in section 7.

2. Related work

Type inference has been a heavily researched topic for the last couple of decades [1, 3, 4, 23, 25, 27–31, 33].

2.1 Static analysis

Most type inference techniques rely only on static information. The first work this field was done by Milner, who described a type inference algorithm named "Algorithm W" [23] that supports polymorphic functions, but not subtyping.

The Cartesian Product Algorithm, known as CPA [1], was developed for dynamically typed language *Self* [33]. This algorithm infers the return type of a method by analyzing the cartesian product of the actual types of the method arguments. CPA is used in the Starkiller [29], a type inferencer and compiler for Python.

RoelTyper is a fast and relatively accurate type inference technique, developed by Pluquet [27]. The algorithm traverses the set of messages sent to a variable together with the assignments to the same variable and infers types for it. This approach was used as a basis for our prototype implementation, as well as for the prototype implementation of the EATI algorithm, which uses the information available in the language ecosystem to increase the available information about types [30]. Since this approach showed a significant improvement when compared to type inference technique that use only static information from the available source code, we used it as a benchmark.

Spoon *et al.,* developed one of the most precise type inference algorithms [31, 32], a *demand-driven* algorithm using *subgoal pruning*. The information used for type inferencing is analyzed on demand, and the precision of the algorithm decreases if some of the subgoals required for type inference need to be discarded, due to complexity reasons.

One of the implemented heuristics to order possible types for a variable is by using information about static class usage [24]. The authors have used statically available information about class name occurrences and object creation to order possible types for variables.

2.2 Dynamic analysis

Some approaches rely on dynamic information for type inference [4, 10, 25]. Such approaches require runnable source code with valid input, either through test cases or symbolic execution [14].

A recent example is dynamic type inference for JavaScript [25], to enable possible type annotations. The authors ran tests to collect dynamic type information used for type inference. The system of interest was instrumented to log types of the variables, which introduced an overhead in the execution.

The other approach for type inference based on dynamic analysis collected by code instrumentation is implemented for Ruby [4]. Even though it is completely based on information collected from program runs, the analysis is sound if program runs used for collecting constraints cover every possible path in the control-flow graph. Thus, in order to have useful type information, the analysis requires running the program with tests. During program runs, variables of interest are wrapped, and monitored when used as receivers or passed as arguments to type-annotated methods. Based on these logs, subtyping constraints are generated for each variable. As the authors state, "the overhead of running the algorithm is currently very high". Our approach collects dynamic type information from the VM, so it does not impose an overhead on the program execution. Dynamic type information has been collected through *any program* run until the present moment, *i.e.,* any system executed within the IDE, and not (only) the system in which types have been inferred. Thus, the approach does not require the system for which types are being inferred to be in a running state.

Static type inference which relies on dynamic collection of data has been developed for Smalltalk [28]. The algorithm infers types of variables based on the values stored in the variables during run-time, and incrementally updates static type information. The main assumption needed for this work is that test coverage is "complete" and that the program of interest is in runnable state. Obviously, the frequently the variable has been encountered during program run, the more precise will be the type information.

3. Gathering of dynamic type information with minimal overhead

3.1 Dynamic type information gathering

We have implemented a dynamic type data gatherer, which collects type information from the virtual machine. The

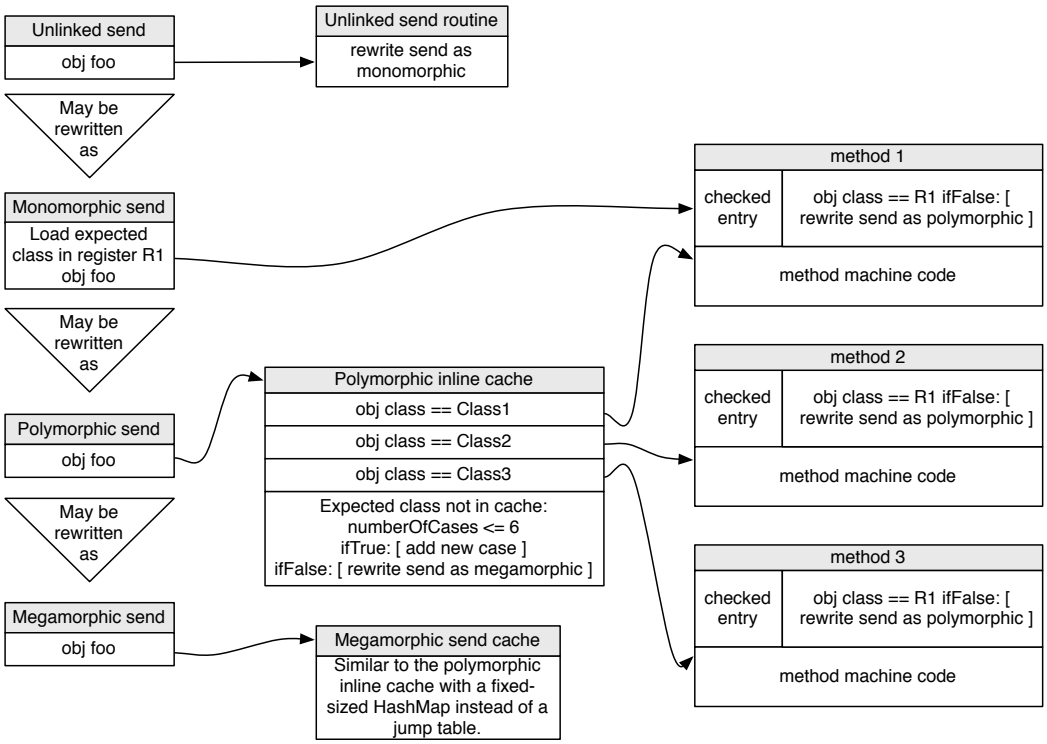

Figure 1. States of the machine code send site for the message send obj foo

gatherer queries the virtual machine for all the message sends that have recently been executed and for each class which has lately been used as a type of a receiver, it calculates the frequency of its usage as a receiver for a message send, and updates the existing information. It is scheduled to run regularly (every 10 seconds) in the Pharo image[1], in order to ensure that collected data is up-to-date. We have chosen the time interval of ten seconds, to ensure that the data is up-to-date, without introducing a major overhead. The final product is the rate of occurrence of each class as a type of the receiver within the used image.

We describe firstly the execution of message sends in the Pharo virtual machine, especially how the inline caches are managed to explain in which circumstances which types are available or not. Then we detail how we extract the cache information and provide it to the Smalltalk runtime.

3.2 Execution of message sends

Bytecode interpretation. Conceptually, the Smalltalk VM executes compiled methods by interpreting the method's bytecodes. Each message send in the bytecode interpreter attempts to fetch the method to execute from the receiver class and the message selector in a global look-up cache, and on cache miss, performs the look-up routine. As the interpreter's cache is global, it is not possible to provide any types from interpreted code.

JIT compilation. In the Pharo VM, when a method is frequently used, if it is eligible for machine code compilation, the JIT translates it to machine code and the VM subsequently uses that version. A method is eligible for machine code compilation if it has a small number of literals specified by the VM (currently 64). A method is detected as frequently used according to two main heuristics:

- The look-up result was already in the global look-up cache for an interpreted send.
- A loop in the method has iterated more than a fixed number of times specified in the VM (currently 20).

In practice, in most cases, frequently used methods are compiled to machine code on the second execution.

Machine code method entries. The Machine code version of methods provides an inline cache for each message send within the method, resulting in better performance. For this optimization, the JIT generates two machine code entries for a method:

- a checked entry: the checked entry ensures that the receiver has a specific class. If this is the case, execution falls through to the unchecked entry, else it calls a relinking routine detailed later.

[1] The term "Pharo image" is used to denote snapshot of the running Pharo system, frozen in time.

• an unchecked entry: this entry executes the method code.

Inline caches. The JIT compiles all method sends as *unlinked sends*. On the first execution, an unlinked send falls back to a routine. This routine finds the method to activate in similar way to the interpreter (global look-up cache search or look-up). If the method is eligible for machine code compilation, and has not already been compiled, it is compiled now. The message send is rewritten to call the checked entry of the method, as shown on the upper side of Figure 1. Roughly 70% of unlinked message sends are rewritten during their lifetime.

A message send calling the checked entry of a method is called a *monomorphic send*. In 90% of the cases, the send site is called on the same receiver type, hence the send site remains monomorphic. However, in 10% of the cases, the send is used on multiple receiver types, and the checked entry of the method called fails. When a method call fails, the send site is rewritten as a call to a jump table. In the jump table, multiple classes are tested in sequence for the receiver type, and when a match is found the unchecked entries of the target methods are called, as shown in the middle of Figure 1.

A send using a jump table is called a *polymorphic send*. The jump table can grow up to a fixed number of cases specified by the VM setting, in our case, six[2]. In 90% of cases, polymorphic sends will keep their jump table representation during their lifetime. However, in the remaining cases, the polymorphic inline cache needs to be rewritten to a hash map search as shown in the bottom of Figure 1.

A send using a local hash map is called a *megamorphic send*. In this case (1% of the used sends), a local hash map with a fixed number of entries is available. The number of entries is specified by the VM setting which is currently eight[3]. Upon execution, the method to activate is searched for in the hash map. On success, the method is activated, else execution falls back to the standard interpreter way of looking up a method (global look-up cache or look-up), and patches the entry in the hash map for the found method, potentially overriding a previous entry.

3.3 Special cases

Code loading. Each time a method is installed or changed, the Smalltalk runtime identifies the method's selector and requests the VM to flush all the inline caches for the given selectors. The VM always completely flush the caches (no partial flushing). Hence type-feedback information during code loading is correct but may be incomplete.

Primitives. For specific primitives such as perform: or withArgs:executeMethod:, the VM does not provide enough runtime type information to infer what method is called. The type-information available is only about the receiver's type, not about the argument values.

Exotic Smalltalk primitives such as become: or adoptInstance: are however fully supported: for the inline cache, only the type of the receiver when the send site is reached during execution matters. If the object's type changes between two sends, each sends will provide type-feedback with different types for the object.

Blocks. As blocks are objects in Pharo, the support for type-feedback on sends with blocks as receiver is complete. Sends inside blocks also provide type-information the same way sends inside methods do.

3.4 Extracting types from inline caches

Information available. If a method is present in the machine code zone, each message send can be in one of four different states:

- unused: 30% of message sends
- monomorphic: 90% of used message sends
- polymorphic: 9% of used message sends
- megamorphic: 1% of used message sends

Due to the cache structure, it is possible to reliably extract a single type from each monomorphic inline cache and from two to six types from each polymorphic inline cache.

Reading the caches. To be able to convert each method activation to a Smalltalk Context instance, to rewrite message sends, and to perform machine code zone garbage collection *etc.*, the JIT annotates each machine code method with a small map describing specific machine instructions. It is possible to walk over the map to access all the message sends present in the machine code zone. This map provides information such as the mapping between the machine code instruction pointer and the bytecode program counter. To read the message send caches of a given method, we iterate over the method's map. For each message send, we analyse machine instructions present to figure out which kind of cache is present, and whether the caches are monomorphic or polymorphic. We read types in the caches, write them into an array, as it is the easiest object to mutate from the VM, and map them to the bytecode program counter. The details of these operations are beyond the scope of this paper.

Providing usable type information. We introduced two new essential primitive methods[4] to obtain type information from the VM as shown on Figure 2. The first primitive, allMachineCodeMethods, answers an array of all methods present in the machine code zone, *i.e.,* all methods with type

[2] The cache size was computed from benchmarks to balance performance and memory overhead.

[3] See footnote 2.

[4] Some messages in the system are responded to primitively. A primitive response is performed directly by the interpreter rather than by evaluating expressions in the method. Essential primitives, in contrast to optional primitives available for performance only, cannot be performed in any other way. For example, Smalltalk without primitives can move values from one variable to another, but cannot add two SmallIntegers together.

information. The second primitive, sendAndBranchData, answers for a given method all the type information available in the inline caches

[5]. For each send site, the type information provided by the primitive consists of an array of types and methods, and the bytecode program counter of the send site. As types are inferred at the AST level, we need to leverage that information. The Opal compiler [6] provides a tool to map the bytecode program counter to AST nodes, which is normally used by the debugger [9] to highlight the code being executed. We used this tool to map the type information from the bytecode program counter to the AST nodes.

```
VirtualMachine>>allMachineCodeMethods
  <primitive: 'primitiveAllMethodsCompiledToMachineCode' module:''>
  ^#()
CompiledMethod>>sendAndBranchData
  <primitive: 'primitiveSistaMethodPICAndCounterData' module:''>
  ^#()
```

Figure 2. New essential primitives

4. Type Inference

To explain the heuristic, we introduce a simple set-theoretic model in Figure 3 that captures key properties for the entities shown in the UML diagram in the Figure 4.

$$msg : V \to \mathcal{P}(S) \quad (1)$$

$$sel : M \to S \quad (2)$$

$$def : M \to C \quad (3)$$

$$sup : C \to C \cup \{null\} \quad (4)$$

$$assign\_types : V \to \mathcal{P}(C) \quad (5)$$

$$under : C \cup \{null\} \times S \to \{true, false\} \quad (6)$$

Figure 3. The core model.

Given a target programming language, C is the domain of all classes, M is the domain of all methods, S is the domain of all selectors. V is the domain of all variables, including instance variables, method arguments and local variables.

Each variable v has a (possibly empty) set of messages $msg(v)$ sent to it in its lexical scope (1). [6]

[5] With specific Just-In-Time compiler settings, in addition to type-feedback, the primitive can provide the values of profiling counters. Counters are installed on some branches in machine code in order to detect frequently executed portions of code and infer basic block usage. That is why the primitive is called sendAndBranchData and not just sendData. In this paper only type-feedback is used so the branch data is not discussed any further.

[6] Note that in this paper we consider the scope for instance variables only to be the methods of the class in which it is defined, but not its subclasses. Considering also the methods of the subclasses to be part of the lexical scope of a variable can only improve the results.

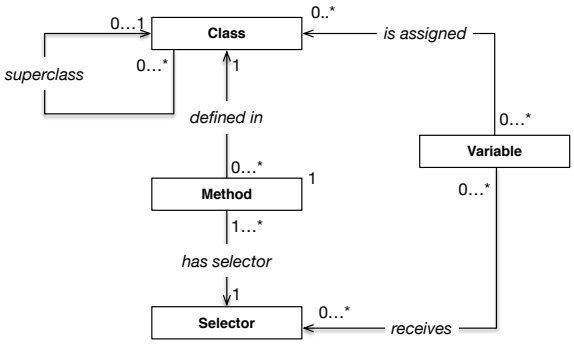

Figure 4. The core model in UML

We call this set of messages the *interface* of the variable v. Each method m has a unique selector $s = sel(m)$ (2), and is defined in a unique class $c = def(m)$ (3). Each class c has a unique superclass $c' = sup(c)$(4). We define the superclass of Object to be *null*, *i.e., sup*(Object) = *null*.

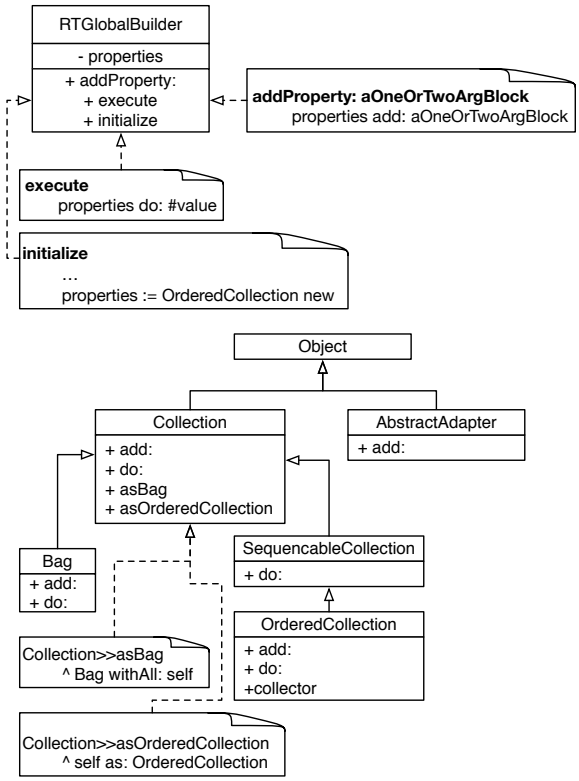

Figure 5. Sample class hierarchy

Consider the example class hierarchy in Figure 5. In this example we see a class RTGlobalBuilder with an instance variable named properties and methods addProperty:, execute and initialize. Within these three methods messages sent to the instance variable properties are

msg(properties) = {add:, do:}

Also, each variable v may have one or more assigned types $c \in$ *assign_types*(v) if the variable v is the left side of an assignment where the right side of the same assignment is a message send to a class which results in creating a new object, *i.e.,* is a call to a constructor (5). Returning to the example, in the method RTGlobalBuilder>>initialize there is an assignment to the instance variable properties of the newly created object of type OrderedCollection, which means that

assign_types(properties) = {OrderedCollection}

Multiple assignments to the same variable are possible, but this is beyond the scope of our small example.

$$\textit{under}(c, s) = s \in \textit{sel}(\textit{def}^{-1}(c)) \vee \textit{under}(\textit{sup}(c), s) \quad (7)$$

$$\textit{intr}(c) = \{s \in S | \textit{under}(c, s) = true\} \quad (8)$$

$$\textit{sel\_types}(v) = \{c \in C | \textit{msg}(v) \subseteq \textit{intr}(c)\} \quad (9)$$

Figure 6. Computing possible types for a variable.

We can now query the model to ascertain the set of possible types for every variable. Each class c can either understand the selector s or not (6). The class c understands selector s if it defines a method $m \in \textit{def}^{-1}(c)$ such that $\textit{sel}(m) = s$ or its superclass *sup*(c) understands it (7), as presented in Figure 6. We also define that *under*(*null*, s) = *false*. The interface of the class c is a set *intr*(c) of all the selectors s that class c understands (8). The class c is a possible type for the variable v if class c understands the interface of the variable v. (9).

It is important to emphasise that our aim is not to provide receiver type information directly from the inline caches, if available, to a developer. This information, available or not in the moment of inferring variable's type, may not be comprehensive. During the lifetime of the image, we collect the information about the frequency of usage of each class as a type of the receiver in the current image. Possible classes for a receiver type, which are inferred due to the message sends and assignments to the variable, are then sorted based on this frequency.

In the example in Figure 5 we see that the class Collection understands the selector add:, as do all of its subclasses, and the class AbstractAdapter, while *under*(RTGlobalBuilder, add:) = *false*.

We can now calculate the interfaces of the classes

intr(Collection) = *intr*(Bag) = *intr*(SequenceableCollection) = {add:, do:, asBag, asOrderedCollection}

intr(OrderedCollection) = {add:, do:, collector, asBag, asOrderedCollection}

intr(AbstractAdapter) = {add:}

Hence, possible types for the variable properties are

sel_types(properties) = {Collection, Bag, SequenceableCollection, OrderedCollection}

4.1 Dynamic information

Let *MS* be the set of all message sends in the target programming language. Each message send has a receiver and a selector sent to the receiver.

$$\textit{dynamic\_type} : \textit{MS} \rightarrow C \quad (10)$$

$$\textit{class\_value}(c) = |\{ms | \textit{dynamic\_type}(ms) = c\}| \quad (11)$$

Figure 7. Calculating class weight.

Each class occurs as the type of a receiver for a message send zero or more times (10). Based on the frequency of the class usage as a receiver type for previously executed message sends during the image lifetime, we calculate the *class_value* (11), as the number of message sends for which this class occurred as a receiver type during run time. *class_value* is a global variable calculated per each class. This information is used to sort the classes that represent possible types for a variable. We extract this information from the virtual machine, with the help of the implemented dynamic type data gatherer.

Dynamically collected information is used to order separately two sets of classes: *assign_types*(v) and *sel_types*(v). In our example in Figure 5, we encounter the following occurrences of the class OrderedCollection, Bag Collection and SequenceableCollection:

class_value(OrderedCollection) = 1487
class_value(Bag) = 34
class_value(Collection) = 1
class_value(SequenceableCollection) = 0

Based on the obtained information, we can now sort the possible types for the variable properties. The list *assign_types*(properties) has only one element, so there is no need for sorting. But the list *sel_types*(properties) has four elements which will be sorted as follows:

1. OrderedCollection
2. Bag
3. Collection
4. SequenceableCollection

4.2 Hierarchy-Based approach

While it is important for the developers to know the precise type of a variable, it is also important to have a notion of the hierarchy of classes to which the run-time type of the variable can belong, since many of the analyzed variables have an interface understood by many independent hierarchies, *i.e.,* hierarchies whose roots do not have a common superclass understanding the same interface. Accordingly, we present two types of information to the developer: the

one obtained by a hierarchy-based approach, the other by a class-based one.

The class-based approach has already been explained throughout section 4.

A variable can have an interface understood by tens, hundreds, or even thousands of classes. Obviously, such information presented to a developer as such is not helpful. We propose to identify the root class in a hierarchy of classes that understand the interface of the variable as a representative for that hierarchy. In the example in Figure 5 the interface of the instance variable properties is understood by four classes belonging to the same hierarchy of classes with root class Collection. We therefore also infer the type of the variable properties as

$$\textit{sel\_types}(\text{properties}) = \{\text{Collection}\}$$

Sets of messages sent to a variable can be understood by multiple hierarchies, but this is beyond our small example.

Let us emphasise here that we do not apply this change to the set *assign_types*(v), but only to the set *sel_types*(v). We consider the set of explicitly assigned types to a variable to be truthful, as it is. The implications of this decision are discussed in section 6.

5. Evaluation

For the purpose of initializing the gatherer, we have run all the tests available in the Pharo image. We have used this information to initialize the gatherer with as much dynamic type information as possible. The collected data has been used to calculate the class value, *class_value*(c) of each class c. For this purpose, we have measured both the time spent to run tests without collecting *class_value* information, and the time spent to run tests during the collection of *class_value* information. The introduced overhead measured around 0.6 milliseconds per method.

For the evaluation purpose, we have used four open-source Pharo projects for which we were able to collect run-time information that closely depicts their real usage: Glamour[7] [8], Roassal2[8] [5], Morphic[15] and Moose[9] [12, 13, 18]. Glamour is a framework for specifying the navigation flow of browsers. Roassal is an agile visualisation engine that graphically renders objects. Morphic is a User Interface construction kit and Moose is a platform for software analysis. These projects provide "example methods" which reflect their real usage: Glamour has 68 of these methods, Morphic 29 and Roassal 948. For Moose we have collected run-time data by performing software analysis on a project. We have loaded an *mse* file which is similar to *xml* file and represents the model of a package. After loading the model, we performed a few queries on it. During the execution of these projects, the information about types of variables was recorded, and this information was declared to be the "ground truth", to which the type information provided by our heuristic was compared. In order to recover these run-time types of the variables, the source code of the projects was instrumented to log the types of the variables as the provided examples were executed. For recovering dynamic type information we have used a tool to track the types of variables at run time, built on top of Reflectivity[10], a reflection API for annotating AST nodes with metalinks.

Dynamic information collected via test runs is used just to initialize the gatherer needed to calculate *class_value*(c), *i.e.,* to order the inferred types of variable. We chose to run all the available tests in the image, in order to initialize the gatherer with as much information as we can. Dynamic type information collected via examples is used just for the evaluation part, as truthful types of variables, to which we compared the statically inferred types.

Types of these variables are then inferred using our heuristic. Examples that we used to collect the run-time information about types covered 114 variables in Glamour, 147 variables in Moose, 563 variables in Morphic and 3935 variables in Roassal2.

We investigated the following research question:

RQ *How well does the proposed heuristic infer type information?*

A summary of the evaluation results is given in Table 1.

5.1 Guessed and "near-guessed" types

If the actual run-time type of a variable has been on the top of the list of sorted types, we label such a variable "guessed". If the variable has n run-time types, where $n > 1$, we consider it to be "guessed" if the set of first n types of the statically inferred list of types is the same as the set of run-time types.

In 59.1% of cases (for 2520 out of 4264 variables in the four projects), the inferred type of the variable is correct, and these variables are considered as "guessed". In these results we omit the variables in the last column, *i.e.,* variables for which we were not able to conclude any other type except Object. We argue that these results could be discarded as they are easily identifiable. This number may seem high, but it is justified by the fast that there are 441 methods already defined in the Object class in Pharo, and it is also possible to add a user-defined method to library classes. These messages can be sent to any Smalltalk object. The heuristic is working slightly better for the types from the standard library, since 1154 of the correctly inferred types are project-related, and 1366 are library types.

If the heuristic failed to promote the correct type to the top of the list, but the correct type is present in the top three, we call these variables "near-guessed". If the variable has n run-time types, where $n > 1$, we consider it as "near-guessed" if the set of first $n+2$ types of the statically inferred

[7] http://www.smalltalkhub.com/#!/~Moose/Glamour

[8] http://smalltalkhub.com/#!/~ObjectProfile/Roassal2

[9] http://www.smalltalkhub.com/#!/~Moose/Moose

[10] http://www.smalltalkhub.com/#!/~RMoD/Reflectivity

Class-based approach							
Project name	#of analyzed variables	#of guessed variables	#of guessed variables - library type	#of guessed variables - package type	#of near-guessed variables	#of incorrectly-guessed variables	#of Object type
Roassal	3935	2165 (60.44%)	1107	1058	222 (6.2%)	1195	353 (33.36%)
Glamour	114	57 (60.04%)	29	28	6 (6.74%)	26 (29.21%)	25
Morphic	563	242 (50.95%)	183	59	44 (9.26%)	189 (39.79%)	88
Moose	147	56 (47.46%)	47	9	19 (16.1%)	43 (36.44%)	29

Hierarchy-based approach							
Project name	#of analyzed variables	#of guessed variables	#of guessed variables - library type	#of guessed variables - package type	#of near-guessed variables	#of incorrectly-guessed variables	#of Object type
Roassal	3935	2566 (71.63%)	1419	1147	466 (13.01%)	550 (15.35%)	353
Glamour	114	79 (88.76%)	35	44	6 (6.74%)	4 (4.49%)	25
Morphic	563	368 (77.47%)	194	174	59 (12.42%)	48 (10.10%)	88
Moose	147	77 (65.25%)	63	14	11 (9.32%)	30 (25.42%)	29

Table 1. Inline caches heuristic

list of types is the superset of the set of run-time types. For example, if the variable has three different run-time types, we consider it to be "near-guessed" if all three types are present in the top five types in the statically inferred list. Our heuristic promoted the correct type to the top three in the list for 291 variables, *i.e.,* 6.82% of variables are "near-guessed".

5.1.1 Incorrectly-guessed types

For 1453 variables the correct type was not in the top three types in the list.

Among these variables, 1190 have an interface not only understood by multiple classes, but also by multiple hierarchies, *i.e.,* classes which do not have a common superclass understanding the required set of messages. The number of these hierarchies ranges from 2 to 204 per variable, with a median of 10 . That is why we deem it is also important to infer the hierarchy to which it may belong, beside the specific type of a variable.

5.1.2 Hierarchy-Based Approach

Figure 8 shows the distribution of the number of possible hierarchies for statically analysed variables. At least half of the variables (50%) have an interface understood by more than four independent hierarchies, and at least 25% of the variables have a possible run-time type which may belong to more than 16 hierarchies.

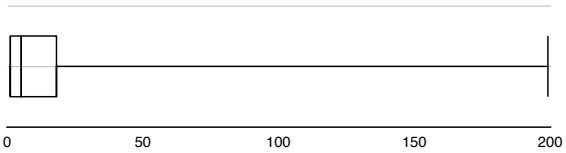

Figure 8. Distribution of the number of possible hierarchies for each statically analysed variable

For 3090, *i.e.,* 72.47% of the variables, the approach correctly inferred the hierarchy of possible types for the variable. The results, labeled *hierarchy-based approach*, are also presented in Table 1. We have again discarded 495 variables for which we were not able to conclude any other type except Object. For 542 variables, *i.e.,* 12.71%, the correct hierarchy was present in the top three in the list.

5.2 Comparison with EATI

We compare our proposed heuristic with EATI, a type inference algorithm that uses information available in the lan-

Type inference	#of analyzed variables	#of guessed variables	#of near-guessed variables
Inline caches heuristic — hierarchy-based	4759	3090	542
EATI	4759	2080	748

Table 2. Comparison with EATI

guage ecosystem [30]. EATI gatheres data from the ecosystem and stores it in a central repository, to be queried when required. EATI shows 95% improvement when compared to a single system type inference techniques, *i.e., RoelTyper* [27], used as its basis. The results of our comparison based on "guessed" and "near-guessed" variables are shown in Table 2.

We have compared the numbers of variables "guessed" by EATI, *i.e.,* variables for which EATI succeeded to promote the correct type to the top of the list, with the number of variables whose type is "guessed" by our technique and the numbers of "near-guessed" variables. Our heuristic infers the correct types of 48.56% more variables. When summed up all the variables for which we have a reasonable precision when inferring types, our heuristic presents an improvement of precision 28.43% of cases. We deem these findings important, since our heuristic yields better results, and they are obtained with considerably less effort and resources.

6. Threats to validity

Even though the proposed should work on every dynamically typed language, we cannot state this without explaratory study.

The main threat to validity comes from collected dynamic information: both data collected through the virtual machine, and used to calculate the class value for each class, and dynamic information declared as "ground truth". For the purpose of evaluation we have tried to collect dynamic information from a wide range of projects, by running tests. The actual dynamic information depends heavily on the ways projects are executed in Pharo image. We have chosen projects *Roassal2*, *Glamour*, *Morphic* and *Moose* to evaluate the heuristics on them, since these projects benefit from the sets of example methods which closely depict their real usage, or we know how to imitate their real usage. It is an open question whether the collected dynamic type information represents true types of variables in these projects. We chose these examples over unit tests, since we feel that they provide more precise information about the usage of the variables than unit tests.

Our choice to treat the assignment types of a variable to be truthful as they are, without considering the subtypes, may have influenced the results. If subtypes would be considered as potential, the results can be influenced in two ways: the number of correctly inferred variables may decrease, due to the increased number of possible types for a variable, or they may increase if the run-time type of a variable is a subtype of the assigned type.

During our evaluation, we have encountered 40 variables whose interface not understood by any single class in the image. We suppose that these 40 variables are so-called "duck-typed" variables. We intend to explore the actual usage of duck-typed variables. These variables can be considered as pollution to any type-inference algorithm.

We have used only intra-procedural analysis in our algorithm. Application of more complex inter-procedural analysis would certainly improve the results.

We take into account neither the usage of dynamic features nor type predicates in Smalltalk. This means that message sends like perform:, become:, isNil are not treated in any special way.

Although our approach imposes minimal overhead, there are a couple drawbacks when it comes to the virtual machine.

We are not able to access all the types that were met at runtime. In practice, frequently used methods and types are always present in the machine code zone with filled caches. However, uncommon types may not be present. Indeed, if a method was executed a single time, it may have been interpreted and hence not provide any type. In addition, the machine code zone has a fixed size. Hence, when the zone is full, the garbage collector frees one quarter of the machine code zone, starting from the least used to the most used methods, losing all the type information present. To solve partially that garbage collection problem, we doubled the size of the machine code zone for our experiments.

Another problem is that the runtime type information provides only a subset of the concrete types. For instance, if a method is present in Collection, and the current code uses it only in OrderedCollection and Set, the type feedback will provide an array with OrderedCollection and Set as types and will not provide the abstract type (Collection) nor the other concrete types, such as Array or Dictionary.

7. Conclusion and future work

Having a type information at compile time can be useful when performing program maintenance tasks. In order to provide accurate information, type inference algorithms are often very complex, and still they suffer from the problem of false positives.

In this paper we have presented a simple heuristic that aims to produce precise type information by using easily accessible information from inline caches. The approach needs no instrumentation, thus imposes minimal execution overhead. The proposed heuristic was assessed by implemented prototype for Pharo Smalltalk. We have focused our attention not only on inferring the correct type of the variable,

but also the correct hierarchy, since more than half of the variables have an interface understood by more than five independent hierarchies.

The proposed heuristic tends to work quite well both for library and project-related types, and produces reasonably correct results. Also, it produces better results than a similar approach, which has more requirements.

We intend to explore possible directions for improving proposed heuristic, *i.e.,* exploring lexical similarities between the variable name and class names.

Acknowledgements

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Analysis" (SNSF project No. 200020-162352, Jan 1, 2016 - Dec. 30, 2018).

This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council, CPER Nord-Pas de Calais/FEDER DATA Advanced data science and technologies 2015-2020.

We thank Eliot Miranda for helping us to implement the primitives we added in the Pharo VM and reviewing all our related commits.

References

[1] O. Agesen. The Cartesian product algorithm. In W. Olthoff, editor, *Proceedings ECOOP '95*, volume 952 of *LNCS*, pages 2–26, Aarhus, Denmark, Aug. 1995. Springer-Verlag.

[2] O. Agesen, J. Palsberg, and M. I. Schwartzbach. Type inference of SELF: Analysis of objects with dynamic and multiple inheritance. In O. Nierstrasz, editor, *Proceedings ECOOP '93*, volume 707 of *LNCS*, pages 247–267, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[3] E. Allende, O. Callaú, J. Fabry, É. Tanter, and M. Denker. Gradual typing for Smalltalk. *Science of Computer Programming*, Aug. 2013.

[4] D. An, A. Chaudhuri, J. Foster, and M. Hicks. Dynamic inference of static types for Ruby. In *Proceedings of the 38th ACM Symposium on Principles of Programming Languages (POPL'11)*, pages 459–472. ACM, 2011.

[5] V. P. Araya, A. Bergel, D. Cassou, S. Ducasse, and J. Laval. Agile visualization with Roassal. In *Deep Into Pharo*, pages 209–239. Square Bracket Associates, Sept. 2013.

[6] C. Bera, S. Ducasse, A. Bergel, D. Cassou, and J. Laval. Handling exceptions. In *Deep Into Pharo*, page 38. Square Bracket Associates, Sept. 2013.

[7] B. Boehm and V. R. Basili. Software defect reduction top 10 list. *Computer*, 34(1):135–137, Jan. 2001.

[8] P. Bunge. Scripting browsers with Glamour. Master's thesis, University of Bern, Apr. 2009.

[9] A. Chiş, T. Gîrba, and O. Nierstrasz. The Moldable Debugger: A framework for developing domain-specific debuggers. In B. Combemale, D. Pearce, O. Barais, and J. J. Vinju, editors, *Software Language Engineering*, volume 8706 of *Lecture Notes in Computer Science*, pages 102–121. Springer International Publishing, 2014.

[10] J. Davies, D. M. Germán, M. W. Godfrey, and A. Hindle. Software bertillonage: Finding the provenance of an entity. In *MSR'11: Proceedings of the 8th International Working Conference on Mining Software Repositories*, pages 183–192, 2011.

[11] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings POPL '84*, Salt Lake City, Utah, Jan. 1984.

[12] S. Ducasse, T. Gîrba, M. Lanza, and S. Demeyer. Moose: a collaborative and extensible reengineering environment. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 55–71. Franco Angeli, Milano, 2005.

[13] S. Ducasse, M. Lanza, and S. Tichelaar. Moose: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems. In *Proceedings of CoSET '00 (2nd International Symposium on Constructing Software Engineering Tools)*, June 2000.

[14] H. Eertink and D. Wolz. Symbolic execution of LOTOS specifications. Memoranda Informatica 91-47, TIOS 91/016, University of Twente, May 1991.

[15] H. Fernandes and S. Stinckwich. Morphic, les interfaces utilisateurs selon squeak, Jan. 2007.

[16] B. Foote and R. E. Johnson. Reflective facilities in Smalltalk-80. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 327–336, Oct. 1989.

[17] J. D. Gannon. An experimental evaluation of data type conventions. *Commun. ACM*, 20(8):584–595, Aug. 1977.

[18] T. Gîrba. The Moose book, 2010.

[19] S. Hanenberg. An experiment about static and dynamic type systems: Doubts about the positive impact of static type systems on development time. *SIGPLAN Not.*, 45(10):22–35, Oct. 2010.

[20] U. Hölzle, C. Chambers, and D. Ungar. *ECOOP'91 European Conference on Object-Oriented Programming: Geneva, Switzerland, July 15–19, 1991 Proceedings*, chapter Optimizing dynamically-typed object-oriented languages with polymorphic inline caches, pages 21–38. Springer Berlin Heidelberg, Berlin, Heidelberg, 1991.

[21] P. Lutz and T. W. F. A controlled experiment to assess the benefits of procedure argument type checking. *IEEE Trans. Softw. Eng.*, 24(4):302–312, Apr. 1998.

[22] C. Mayer, S. Hanenberg, R. Robbes, E. Tanter, and A. Stefik. An empirical study of the influence of static type systems on the usability of undocumented software. *SIGPLAN Not.*, 47(10):683–702, Oct. 2012.

[23] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[24] N. Milojković and O. Nierstrasz. Exploring cheap type inference heuristics in dynamically typed languages. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2016. To Appear.

[25] M. P. Odgaard. Javascript type inference using dynamic analysis. Master's thesis, Aarhus University, 2014.

[26] J. K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31(3):23–30, Mar. 1998.

[27] F. Pluquet, A. Marot, and R. Wuyts. Fast type reconstruction for dynamically typed programming languages. In *DLS '09: Proceedings of the 5th symposium on Dynamic languages*, pages 69–78, New York, NY, USA, 2009. ACM.

[28] P. Rapicault, M. Blay-Fornarino, S. Ducasse, and A.-M. Dery. Dynamic type inference to support object-oriented reengineering in Smalltalk, 1998. Proceedings of the ECOOP '98 International Workshop Experiences in Object-Oriented Reengineering, abstract in Object-Oriented Technology (ECOOP '98 Workshop Reader forthcoming LNCS).

[29] M. Salib. Faster than C: Static type inference with Starkiller. In *in PyCon Proceedings, Washington DC*, pages 2–26. SpringerVerlag, 2004.

[30] B. Spasojević, M. Lungu, and O. Nierstrasz. Mining the ecosystem to improve type inference for dynamically typed languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! '14, pages 133–142, New York, NY, USA, 2014. ACM.

[31] S. A. Spoon and O. Shivers. Demand-driven type inference with subgoal pruning: Trading precision for scalability. In *Proceedings of ECOOP'04*, pages 51–74, 2004.

[32] S. A. Spoon and O. Shivers. Dynamic data polyvariance using source-tagged classes. In R. Wuyts, editor, *Proceedings of the Dynamic Languages Symposium'05*, pages 35–48. ACM Digital Library, 2005.

[33] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 227–242, Dec. 1987.